N15-ALT.

N15-P

[illegible]

61-TM

277394

P.12

MOVEMENT OF IMPLICIT PARALLEL AND VECTOR EXPRESSIONS OUT OF PROGRAM LOOPS

Paul B. Schneck

Institute for Space Studies
New York, N. Y. 10025

ABSTRACT

The concept of "interference" is introduced and related to the problem of transforming implicit parallel and vector operations to an explicit form. Statement types and constructs which cause interference are identified. Programming techniques which avoid use of potentially interfering statements result in programs which are better suited for operation on parallel and vector processors.

The analysis of the inhibiting effect of interfering statements on the motion of parallel and vector expressions has been implemented and demonstrated as an addition to an existing compiler which recognizes parallelism implicit in serial programs.

MOVEMENT OF IMPLICIT PARALLEL AND VECTOR EXPRESSIONS OUT OF PROGRAM LOOPS

INTRODUCTION

This paper discusses the ability to move statements or expressions out of the range of a program loop so that serial operations may be replaced by parallel or vector* operations. An earlier paper (Schneck, 1972) described the recognition of implicit parallel expressions in a high level language. This paper deals with the treatment of such expressions after recognition. We define the conditions which permit or inhibit the movement of potentially parallel expressions out of program loops.

MOTION OF EXPRESSIONS

The ideal disposition of a parallel expression within a program loop is removal and execution outside the loop. If there were no "interference" all such expressions could be moved in front of the loop and their results used within the loop. Figure 1a is an example of a program loop which contains no statements interfering with the motion of parallel statements. All parallel statements are removed from the loop as shown in Figure 1b. If interfering statements are

* For the remainder of this paper we will use the term "parallel" in place of the bulky "parallel or vector."

present the situation gets quite complex. When one statement cannot be moved it may prevent another from being moved. The next sections examine relationships between statements and their effect on the "moveability" of a statement.

DIRECTION OF MOTION

The example given above illustrated upward motion of parallel expressions. That is, a parallel expression was moved to precede the loop in which it originally appeared. This is the most natural formulation in the context of a (locally) first-to-last sequence of statement analysis: statement ordering is preserced without regard to any potential interference.*

A statement which cannot be moved (upward) to precede a loop is examined to determine whether it can be moved (downward) to follow the loop. An interfering statement acts as a barrier to the passage of certain variables. Figure 2 illustrates forms of interference which may occur. The important point is that regardless of the configuration of interfering statements a program loop will be divided into at most three regions: first, a region permitting only upward motion; second a region surrounded by (the extreme) interfering statements and not permitting any motion; third a region permitting only downward motion. As shown, one or two of these regions may be null.

*Interference, which has still not been defined is a function of the variables. A statement may interfere with only some, but not all, variables.

INTERFERENCE AND MOVEMENT

An expression which cannot be moved is said to be an interfering expression. There are three types of interfering expressions:

1. An expression in a non-arithmetic statement (e.g. READ, CALL, etc.) which must be performed in a loop.
2. An expression which is not a function solely of constants, relative constants, and parallel operands (e.g. A(I)*J).
3. Any expression which may not be performed in parallel -- for the above reasons, or because recursive relationships will not permit parallel operation (e.g. A(I+1)=A(I)+A(I-1)).

An interfering expression may limit the movement of otherwise moveable expressions. Figure 3 details an example of the effects of interfering expressions.

An interfering definition is a barrier to the movement of both uses and definitions. Any uses moved across a definition will employ the wrong (old) values. Definitions moved across a definition will be nullified by the one they originally followed.

An interfering use is also a barrier to the movement of definitions. Definitions moved across a use will result in the premature replacement and use of old values by new. However, interfering uses do not inhibit the movement of other uses -- there is no conflict between use and definition in this case.

The relationship between interfering statements and other statements may be summarized in tabular form:

Interfering Variable

Target Variable		REFERENCE	DEFINITION
	REFERENCE	MOVE	INHIBIT
	DEFINITION	INHIBIT	INHIBIT

IMPLEMENTATION: INTERFERENCE NUMBERS

A pair of "interference numbers", updated as successive statements are processed, describe the instantaneous moveability of each variable. The numbers hold a variable's status with respect to references and definitions. A reexamination of Figure 2 will show that there are only three possible configurations involving the use of a variable and interfering expressions:

α) The variable precedes any interfering expressions.

β) The variable follows all interfering expressions.

γ) The variable is between interfering expressions.

The first configuration includes the case in which there are no interfering statements.

Because of the need to determine the extreme interfering expressions for a variable, two passes over the text are required. All interference numbers are initialized to zero. During the first pass, which is necessary for optimization and information gathering, the definition and/or reference interference numbers are set to the statement position whenever an interfering expression is encountered. If the interfering expression is a definition, which can inhibit the movement of references and definitions, both interference numbers will be set. If the interfering expression is a reference, which can only inhibit the movement of definitions, then only the reference interference number is set. Thus, at the end of the first pass the interference numbers indicate the position of the last interfering expression for each variable.

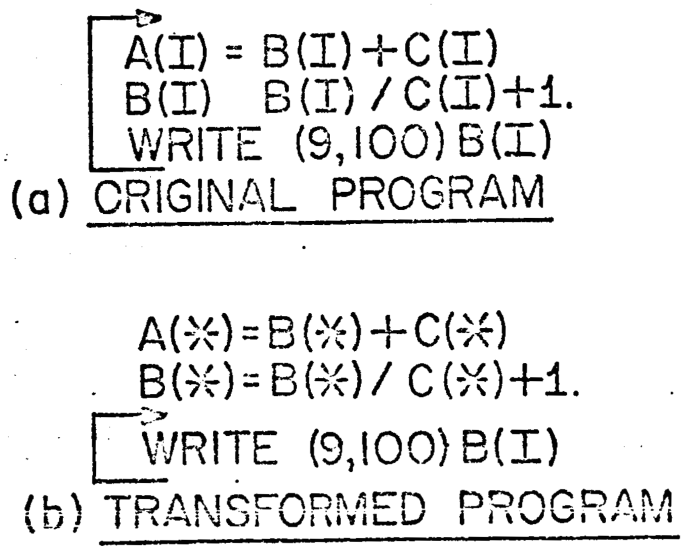
An interference number which remains at zero indicates that no interfering expressions are present. Those interference numbers which are not zero are adjusted to indicate the statement at which downward motion becomes permissible. When the parallel nature of a variable is destroyed it cannot be recovered until a later point where all paths affecting the variable have rejoined as illustrated in Figure 4. During the second pass, when output code is generated, variables will be moved up to permit parallel operations unless flagged otherwise. Flagging of the interference number components occurs when a variable is used (referenced or defined) in an interfering expression. The flag indicates that no motion is currently possible. When the statement indicated by the interference number is reached the flag will be reset to indicate that downward motion is possible. In cases of conflict, e.g. where the reference flag indicates upward motion is possible and the definition-flag indicates downward motion, both conditions must be satisfied; in this instance there will be no motion.

CONCLUSION

Conditions affecting the movement of potentially parallel expressions out of program loops are described, and the results of a working model of this treatment are illustrated. The constructs inhibiting movement of parallel expressions may be related to current programming techniques in an effort to minimize situations where they occur.

REFERENCE

. Schneck, P. B., "Automatic Recognition of Parallel/Vector Operations in a Higher Level Language", Proceedings ACM 1972 National Conference.



```
A(I) = B(I)+C(I)
B(I)   B(I) / C(I)+1.
WRITE  (9,100) B(I)
```

(a) ORIGINAL PROGRAM

```
A(*)=B(*)+C(*)
B(*)=B(*)/ C(*)+1.
WRITE  (9,100) B(I)
```

(b) TRANSFORMED PROGRAM

Figure 1 - Movement of Expressions With No Interference

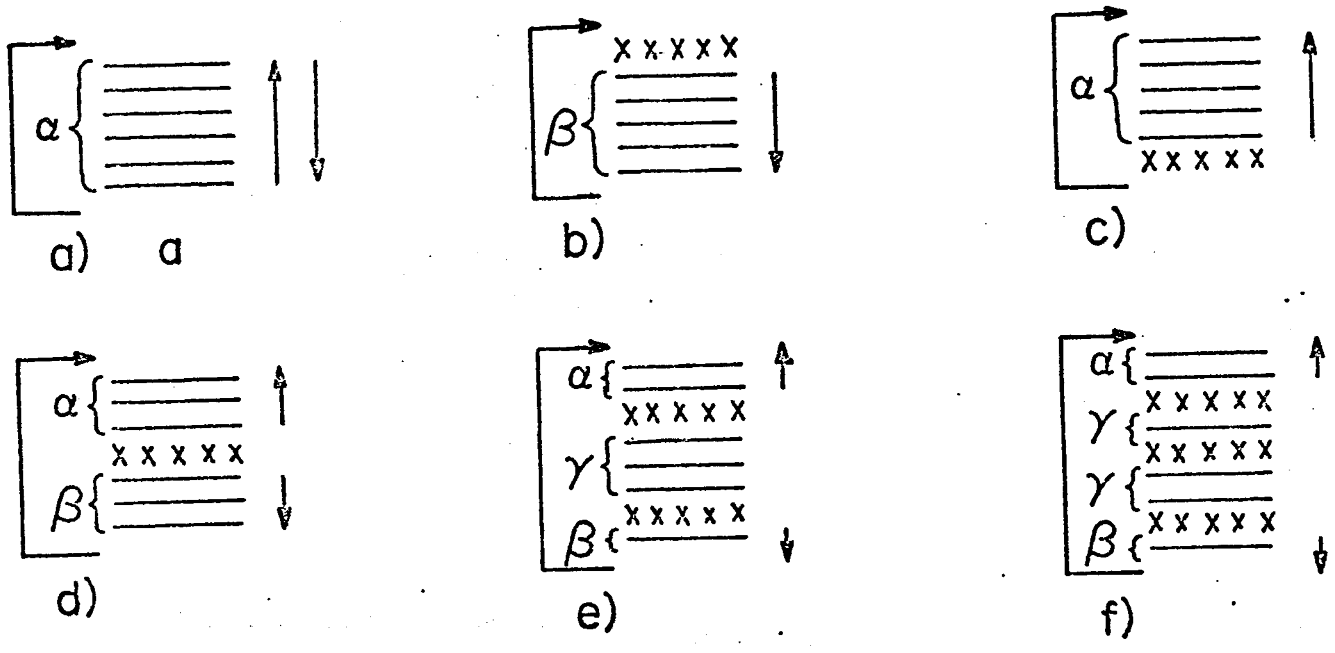
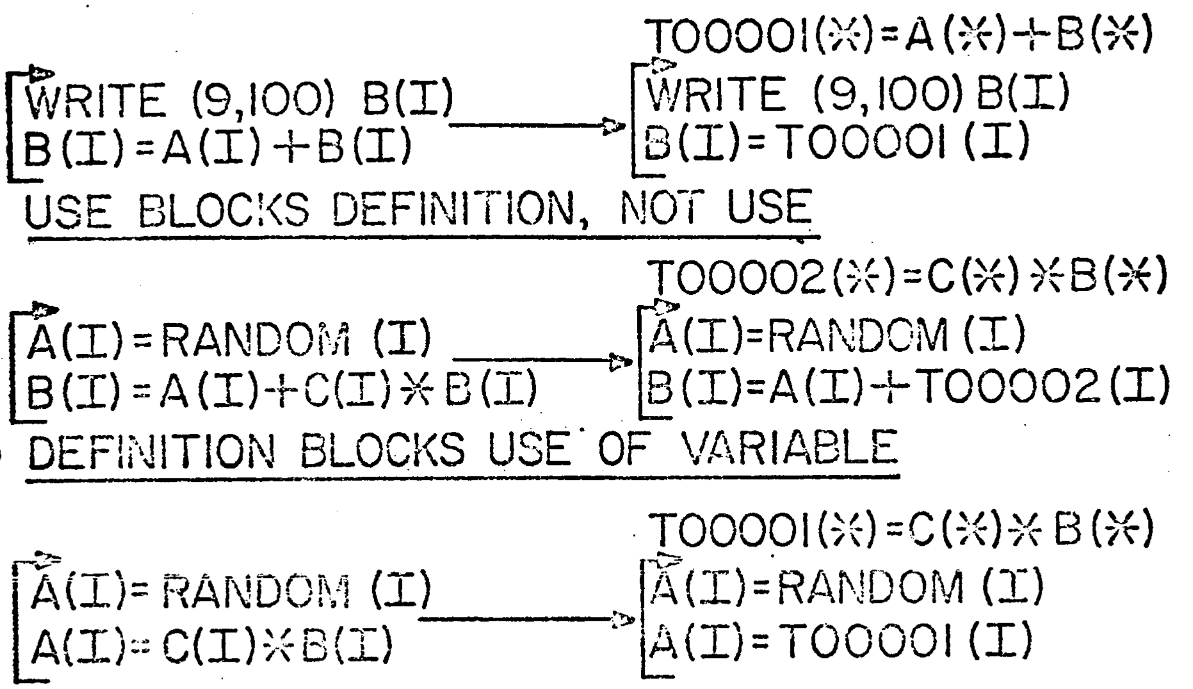


Diagram	Nature of Interference	Result
a.	None	upward or downward motion
b.	Upward	downward motion
c.	Downward	upward motion
d.	Central	upward motion from above, downward from below
e.	Multiple (2)	upward motion from above, downward from below
f.	Multiple (N)	upward motion from above highest, downward from below lowest

Figure 2. Forms of Interference



```
                                    T00001(*)=A(*)+B(*)
WRITE (9,100) B(I)              --> WRITE (9,100) B(I)
B(I)=A(I)+B(I)                      B(I)=T00001(I)
```

(a) USE BLOCKS DEFINITION, NOT USE

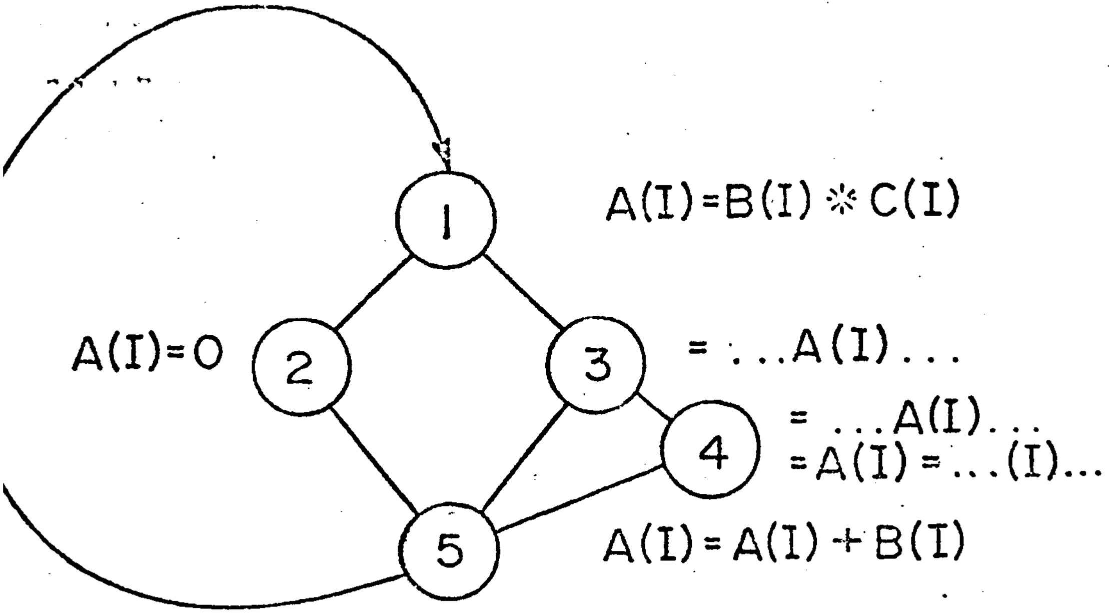
```
                                    T00002(*)=C(*)*B(*)
A(I)=RANDOM (I)                 --> A(I)=RANDOM (I)
B(I)=A(I)+C(I)*B(I)                 B(I)=A(I)+T00002(I)
```

(b) DEFINITION BLOCKS USE OF VARIABLE

```
                                    T00001(*)=C(*)*B(*)
A(I)=RANDOM (I)                 --> A(I)=RANDOM (I)
A(I)=C(I)*B(I)                      A(I)=T00001(I)
```

(c) DEFINITION BLOCKS DEFINITION, NOT OTHER USES

Figure 3-Types of Interference



a) FLOWCHART

Statement	Position	Block
DO 9 I = 1,100	1	1
A(I)=B(I) * C(I)	2	
IF (A(I) ...) GO TO 3	3	
A(I)=0.	4	2
GO TO 5	5	
3 = ... A(I) ...	6	3
IF(B(I) ...) GO TO 5	7	
= ... A(I)	8	4
A(I)= ...	9	
5 A(I)=A(I)+B(I)	10	5
9 CONTINUE	11	

b) PROGRAM SEGMENT

Position	Pass 1	Adjustment	Pass 2	
	reference/definition		up/down	– motion
1-3	0/0	10/10	✓/X	UP
4-8	4/4		X/X	NONE
9-11	9/9		X/✓	DOWN

c) FLAGS FOR THE VARIABLE "A"

Figure 4. Flow and the Interference Numbers